IJoL International Journal of Librarianship

ISSN: 2474-3542 Journal homepage: https://journal.calaijol.org

Implementing Retrieval-Augmented Generation for Academic Libraries: A Technical Case Study using Azure AI

Wei Xuan

Abstract:

This article details the technical development of a Retrieval-Augmented Generation (RAG) system designed to enhance discovery within an academic library's institutional repository. Conducted during a six-month research leave in 2025, this project explores the practical application of emerging cloud-based AI tools in a library context. We developed a prototype that integrates the University of Manitoba's MSpace repository with Microsoft Azure AI services. The system utilizes an OAI-PMH harvester to retrieve metadata, generates semantic vector embeddings via the text-embedding-ada-002 model, and indexes these vectors in Azure AI Search. A custom front-end application facilitates both traditional keyword search and generative, context-aware chat interactions. This paper documents the development environment, script logic, and specific technical challenges overcome—such as OAI-PMH pagination errors and API versioning conflicts—providing a reproducible roadmap for libraries seeking to explore semantic search technologies.

To cite this article:

Xuan, W. (2026). Implementing Retrieval-Augmented Generation for Academic Libraries: A Technical Case Study using Azure AI. *International Journal of Librarianship*, *11*(1), 151-166. https://doi.org/10.23974/ijol.2026.vol11.1.601

INTERNATIONAL JOURNAL OF LIBRARIANSHIP, 11(1), 151-166

ISSN: 2474-3542

Implementing Retrieval-Augmented Generation for Academic Libraries: A Technical Case Study using Azure AI

Wei Xuan, University of Manitoba, Canada

ABSTRACT

This article details the technical development of a Retrieval-Augmented Generation (RAG) system designed to enhance discovery within an academic library's institutional repository. Conducted during a six-month research leave in 2025, this project explores the practical application of emerging cloud-based AI tools in a library context. We developed a prototype that integrates the University of Manitoba’s MSpace repository with Microsoft Azure AI services. The system utilizes an OAI-PMH harvester to retrieve metadata, generates semantic vector embeddings via the text-embedding-ada-002 model, and indexes these vectors in Azure AI Search. A custom front-end application facilitates both traditional keyword search and generative, context-aware chat interactions. This paper documents the development environment, script logic, and specific technical challenges overcome—such as OAI-PMH pagination errors and API versioning conflicts—providing a reproducible roadmap for libraries seeking to explore semantic search technologies.

Keywords: Retrieval-Augmented Generation (RAG), Institutional Repositories, Semantic Search, Azure AI, DSpace

INTRODUCTION

Large Language Models (LLMs) have established themselves as powerful computational tools capable of understanding and producing human language with remarkable proficiency. Their versatility allows them to execute a wide range of Natural Language Processing (NLP) tasks, ranging from translation to complex question answering. The architectural foundation of these models is the transformer, introduced by Vaswani et al. in 2017, which utilizes self-attention mechanisms to significantly enhance language modeling capabilities. Driven by continuous improvements in design and training methodologies, LLMs process vast datasets to identify intricate linguistic patterns and generate contextually appropriate text. Despite their capabilities, LLMs possess inherent limitations that can hinder their utility in academic and professional settings due to their reliance on static training data.

To mitigate these issues, the field has pivoted toward Retrieval-Augmented Generation (RAG). This hybrid architecture represents a paradigm shift in NLP by moving away from a sole reliance on pre-trained, static knowledge. Instead, RAG systems integrate a retrieval mechanism that dynamically accesses external knowledge sources during the generation process, thereby enhancing the model's contextual understanding. By grounding outputs in external data, RAG allows LLMs to adapt to new information without the need for constant retraining, ensuring efficiency and applicability across diverse domains.

This project, conducted during a six-month research leave in 2025, leverages these advancements to prototype a library-specific RAG system using Microsoft Azure, aiming to bridge the gap between static library metadata and dynamic, AI-driven user inquiry.

LITERATURE REVIEW

The application of Retrieval-Augmented Generation (RAG) and Generative AI (GAI) is expanding rapidly across various disciplines, offering distinct advantages over standalone LLMs. By integrating external literature and databases, these systems address the temporal limitations of pre-training and provide domain-specific accuracy.

RAG Applications Beyond the Library

In the healthcare sector, RAG frameworks have been instrumental in assisting clinicians by grounding AI responses in the latest medical literature (Finkelstein et al., 2024; Wang et al., 2024). For instance, systems utilizing RAG can detect major health events in electronic records (Kaplinsky et al., 2024) and support patient education by ensuring answers reflect current medical advancements (Wang et al., 2024). Similarly, advancements in the biomedical field, such as the DRAGON-AI project, demonstrate how dynamic retrieval can optimize decision-making processes for complex ontologies (Toro et al., 2023).

In the realm of governance and social philosophy, RAG models facilitate better citizen-government interaction. Yun et al. (2024) highlight how these systems can retrieve specific policy documents to generate clear explanations for the public, thereby improving transparency and bridging understanding gaps in public administration. Furthermore, the utility of retrieval augmentation extends to information verification. Ni et al. (2024) found that LLMs equipped with retrieval capabilities were effective in fact-checking health-related claims by focusing on context specificity, which is crucial for safe information dissemination.

AI and RAG in the Library Information Ecosystem

Within the library and information science landscape, the focus has shifted toward enhancing user experience (Meakin, 2024; Rahman & Islam, 2024) , search functionality (Columbia University, 2024) , and professional competency through AI (Kautonen & Gasparini, 2024). Unlike generic voice assistants, generative tools in libraries require careful implementation to ensure reliability. Wheatley and Hervieux (2024) compared ChatGPT with traditional assistants like Alexa and Siri, noting that while GAI could provide more relevant answers to reference queries, the inconsistency of its accuracy necessitated significant librarian oversight.

Innovative platforms are also emerging to support academic workflows. Alshammari et al. (2024) developed "PyZoBot," a conversational platform that synthesizes information from curated Zotero reference libraries using advanced RAG techniques, demonstrating how AI can streamline information extraction for researchers. Similarly, initiatives like CORE-GPT highlight the potential of integrating reliable Open Access research into LLMs to ensure credible and trustworthy question answering (Pride et al., 2023).

The integration of these tools also demands new frameworks for staff training and partnership. Kautonen and Gasparini (2024) proposed the "B-Wheel" model to build AI competencies, emphasizing design thinking and hands-on learning to prepare library teams for generative technologies. Collaboration is equally vital; Kassorla et al. (2024) argue that partnerships between librarians and technologists can drive user-centric innovation and improve AI literacy initiatives.

Finally, AI is reshaping discovery through personalization. Meakin (2024) explored how GAI can analyze user behavior to suggest relevant resources, effectively streamlining the research process. However, the study cautioned against an overreliance on algorithmic recommendations, which could potentially limit the diversity of resources users encounter. By integrating RAG into library search tools, institutions can offer advanced filtering and personalized recommendations, making resources more accessible while maintaining the integrity of the discovery process.

This project addresses a specific gap in the implementation of semantic search technologies within institutional repositories. While commercial tools and general chatbot interfaces are becoming common, there is a distinct need for technical case studies that detail the end-to-end development of RAG systems tailored to the specific metadata structures of academic libraries. By prototyping a system that integrates the University of Manitoba’s MSpace repository with cloud-based vector search, this work aims to demonstrate how libraries can move beyond keyword-based discovery. It provides a practical framework for leveraging RAG to make open scholarship more discoverable, intuitive, and conversationally accessible, thereby aligning technical innovation with the core library mission of equitable access to information

APPLICATION DEVELOPMENT

Development Environment

The application was developed in a Python 3.9+ environment using a modular architecture. The core infrastructure relies on **Microsoft Azure**, specifically:

- **Azure OpenAI Service:** Hosted the gpt-35-turbo-16k model for chat completion and text-embedding-ada-002 for vector generation.
- **Azure AI Search:** Served as the vector store and retrieval engine using the 2024-07-01 API version. The service was configured at the Standard tier to support semantic search capabilities.

- **Local Client:** A MacBook terminal environment using venv for dependency management. Key libraries included sickle for OAI-PMH harvesting, streamlit for the user interface, and the azure-search-documents SDK for indexing.

Note on Model Availability: This project was implemented in 2025. The specific models and API versions documented here—such as gpt-35-turbo-16k and text-embedding-ada-002—reflect the Azure AI ecosystem at that time. Given the rapid pace of AI development, these specific deployments may be deprecated, renamed, or superseded by newer iterations in future implementation contexts.

Phase 1: Data Harvesting

The first challenge was retrieving metadata from the MSpace repository. Initial attempts using standard OAI-PMH harvesters failed due to server-side timeouts (HTTP 500 errors) when requesting large sets, such as the *Faculty of Graduate Studies* collection.

To resolve this, we developed a robust harvesting script (harvest_oai_simple.py) that utilizes a "ListIdentifiers then GetRecord" strategy. Instead of requesting massive pages of full records, the script first retrieves lightweight headers. It then iterates through identifiers to fetch full records individually. This granular approach, combined with retry logic, successfully retrieved a large set of records from the server. The output was standardized into a JSON Lines (.jsonl) format, capturing Dublin Core fields like title, creator, subject, and description.

Phase 2: Vectorization and Processing

Once harvested, the textual metadata needed to be converted into a machine-readable format. We utilized the text-embedding-ada-002 model to generate 1536-dimensional vector embeddings. The script (generate_embeddings.py) concatenates key fields (Title, Subject, Description) into a single text block for embedding.

A critical data processing step involved the document keys. Azure AI Search has strict character restrictions for document keys (allowing only letters, digits, dashes, and underscores). The raw OAI identifiers (e.g., oai:mspace.lib.umanitoba.ca:1993/38371) contained invalid characters like colons and slashes. We implemented a Base64 URL-safe encoding step in the upload process to ensure every document had a valid, reversible key.

Phase 3: Indexing and Vector Search Configuration

Configuring the Azure AI Search index required precise schema definitions to support vector search.

The index creation script (recreate_index.py) defines a schema that includes:

1. **Searchable Text Fields:** For traditional keyword search (Title, Creator, etc.).
2. **Vector Field:** A field named embedding typed as Collection(Edm.Single) with 1536 dimensions.

3. **Vector Profile:** Configuration for the HNSW (Hierarchical Navigable Small World) algorithm, which enables efficient approximate nearest neighbor search.

Phase 4: The Streamlit User Interface

The final user application (streamlit_app.py) integrates two distinct search modalities:

1. **Search Tab:** Executes a standard REST API call to Azure Search using keyword matching. It decodes the Base64 identifiers to present clickable links back to the MSpace repository.
2. **Chatbot Tab (RAG):** Implements the full RAG pipeline. When a user asks a question, the app:
    - Generates an embedding for the user's prompt via Azure OpenAI.
    - Sends a vectorQueries payload to Azure AI Search to retrieve the top 3 semantically similar records.
    - Constructs a system prompt containing these metadata records as "context".
    - Sends the context and user question to GPT-3.5, producing an answer grounded in the library's collection.

CONCLUSION

This development work, conducted during a research leave in 2025, demonstrates the technical viability of building a custom RAG system for academic libraries using off-the-shelf cloud components. By successfully overcoming challenges related to legacy data harvesting (OAI-PMH) and modern vector search configurations (Azure API), we have created a prototype that enhances the discoverability of open scholarship. The resulting application offers a dual interface that satisfies both traditional search behaviors and the growing demand for conversational, AI-driven research assistance. Future work will focus on scaling the index to include full-text documents and conducting user studies to evaluate result relevance.

References

Alshammari, S., Basalelah, L., Walaa Abu Rukbah, Alsuhibani, A., & Wijesinghe, D. S. (2024). PyZoBot: A Platform for Conversational Information Extraction and Synthesis from Curated Zotero Reference Libraries through Advanced Retrieval-Augmented Generation. *arXiv.Org*. https://doi.org/10.48550/arXiv.2405.07963

Burtsev, M., Reeves, M., & Job, A. (2024). The Working Limitations of Large Language Models. *MIT Sloan Management Review*, *65*(2), 8–10.

Columbia University. (2024, May 24). Enhancing library search system with AI technology at Columbia University. https://etc.cuit.columbia.edu/news/AICoP-library-augment-discovery-with-AI

Finkelstein, J., Moskovitch, R., & Parimbelli, E. (2024). *Artificial Intelligence in Medicine: 22nd International Conference, AIME 2024, Salt Lake City, UT, USA, July 9-12, 2024, Proceedings, Part I* (2024th edition, Vol. 14844). Springer.

Huang, Y., & Huang, J. (2024). A survey on retrieval-augmented text generation for large language models. *arXiv.Org.* http://arxiv.org/abs/2404.10981

Kamath, U., Keenan, K., Somers, G., & Sorenson, S. (2024). *Large language models : a deep dive : bridging theory and practice* (2024th edition). Springer. https://doi.org/10.1007/978-3-031-65647-7

Kaplinsky, P., Singh, R., Fusillo, T. F., Leader, A., Zwicker, J. I., & Mantha, S. (2024). Retrieval augmented generation for the detection of major bleeding events in the electronic health record. Blood, 144(Supplement 1), 2263. https://doi.org/10.1182/blood-2024-203911

Kassorla, M., Georgieva, M., & Papini, A. (2024). AI literacy in teaching and learning: A durable framework for higher education. Educause. https://www.educause.edu/content/2024/ai-literacy-in-teaching-and-learning/introduction

Kautonen, H., & Gasparini, A. A. (2024). B-Wheel – Building AI competences in academic libraries. *The Journal of Academic Librarianship*, *50*(4), Article 102886. https://doi.org/10.1016/j.acalib.2024.102886

Meakin, L. (2024). Exploring the Impact of Generative Artificial Intelligence on Higher Education Students' Utilization of Library Resources: A Critical Examination. *Information Technology and Libraries*, *43*(3), Article 17246. https://doi.org/10.5860/ital.v43i3.17246

Ni, Z., Qian, Y., Chen, S., Jaulent, M.-C., & Bousquet, C. (2024). Scientific evidence and specific context: leveraging large language models for health fact-checking. *Online Information Review*, *48*(7), 1488–1514. https://doi.org/10.1108/OIR-02-2024-0111

Pride, D., Cancellieri, M., & Knoth, P. (2023). CORE-GPT: Combining open access research and large language models for credible, trustworthy question answering. *arXiv.Org* http://arxiv.org/abs/2307.04683

Rahman, M. H., & Islam, M. N. (2024). The Impact of ChatGPT for Enhancing Knowledge Management in University Libraries. *Journal of Web Librarianship*, *18*(4), 177–196. https://doi.org/10.1080/19322909.2024.2391907

Toro, S., Anagnostopoulos, A. V., Bello, S., Blumberg, K., Cameron, R., Carmody, L., Diehl, A. D., Dooley, D., Duncan, W., Fey, P., Gaudet, P., Harris, N. L., Joachimiak, M., Kiani, L., Lubiana, T., Munoz-Torres, M. C., O'Neil, S., Osumi-Sutherland, D., Puig, A., … Mungall, C. J. (2024). Dynamic Retrieval Augmented Generation of Ontologies using Artificial Intelligence (DRAGON-AI). *arXiv.Org*. https://doi.org/10.48550/arxiv.2312.10904

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need. *arXiv.Org*, https://doi.org/10.48550/arXiv.1706.03762

Wang, C., Ong, J., Wang, C., Ong, H., Cheng, R., & Ong, D. (2024). Potential for GPT Technology to Optimize Future Clinical Decision-Making Using Retrieval-Augmented Generation. *Annals of Biomedical Engineering*, *52*(5), 1115–1118. https://doi.org/10.1007/s10439-023-03327-6

Wheatley, A., & Hervieux, S. (2024). Comparing generative artificial intelligence tools to voice assistants using reference interactions. *The Journal of Academic Librarianship*, *50*(5), Article 102942. https://doi.org/10.1016/j.acalib.2024.102942

Yun, L., Yun, S., & Xue, H. (2024). Improving citizen-government interactions with generative artificial intelligence: Novel human-computer interaction strategies for policy understanding through large language models. *PloS One*, *19*(12), Article e0311410. https://doi.org/10.1371/journal.pone.0311410

Annexes: Application Scripts

Annex A: OAI-PMH Harvester (harvest_oai_simple.py)

```
#!/usr/bin/env python3
import argparse
import json
import time
from typing import Any, Dict, List
from sickle import Sickle
from sickle.models import Record, Header
from requests.exceptions import HTTPError, RequestException

def first_or_empty(v) -> str:
    """Helper to safely extract the first item from a list or return a string."""
    if isinstance(v, list):
        return v[0] if v else ""
    return v or ""

def join_list(v) -> str:
    """Helper to join list elements into a single string."""
    if isinstance(v, list):
        return "; ".join([str(x) for x in v if x])
    return v or ""

def dc_record_to_row(rec: Record) -> Dict[str, Any]:
    """Parses a Sickle record into a flat dictionary suitable for JSON export."""
    md = rec.metadata or {}
    row = {
        "oai_identifier": getattr(rec.header, "identifier", ""),
        "identifier": first_or_empty(md.get("identifier")),
        "title": first_or_empty(md.get("title")),
        "creator": join_list(md.get("creator", [])),
        "subject": join_list(md.get("subject", [])),
        "description": first_or_empty(md.get("description")),
        "date": first_or_empty(md.get("date")),
        "type": first_or_empty(md.get("type")),
        "format": first_or_empty(md.get("format")),
        "language": first_or_empty(md.get("language")),
        "source": first_or_empty(md.get("source")),
    }

    # Logic to derive a handle URL from the identifier if possible
    handle_candidate = row["identifier"] or row["oai_identifier"]
    if ":" in handle_candidate:
        tail = handle_candidate.split(":")[-1]
        if "/" in tail:
            row["handle"] = tail
            row["link"] = f"https://mspace.lib.umanitoba.ca/handle/{tail}"
    return row

def fetch_with_retries(sickle: Sickle, oid: str, prefix: str, retries: int, delay: float) -> Record |
None:
    """Fetches a single record with retry logic for robust harvesting."""
    for attempt in range(1, retries + 1):
        try:
            rec = sickle.GetRecord(identifier=oid, metadataPrefix=prefix)
            if getattr(rec, "deleted", False):
                return None
            return rec
        except (HTTPError, RequestException) as e:
            if attempt == retries:
                print(f"  [Error] Failed to fetch {oid}: {e}")
                return None
            time.sleep(delay * attempt)
        except Exception as e:
            print(f"  [Error] Unexpected error for {oid}: {e}")
            return None
    return None

def main():
    parser = argparse.ArgumentParser(description="Robust OAI-PMH Harvester")
    parser.add_argument("--endpoint", default="https://mspace.lib.umanitoba.ca/oai/request")
    parser.add_argument("--set", required=True, help="SetSpec to harvest")
    parser.add_argument("--out", default="harvest.jsonl", help="Output file")
    args = parser.parse_args()

    sickle = Sickle(args.endpoint, timeout=60, retry_backoff_factor=2)

    print(f"Fetching identifiers for set: {args.set}...")
    try:
        headers = sickle.ListIdentifiers(metadataPrefix="oai_dc", set=args.set)
        ids = [getattr(h, "identifier", "") for h in headers if getattr(h, "status", None) != "deleted"]
    except Exception as e:
        print(f"Error listing identifiers: {e}")
        return

    print(f"Found {len(ids)} records. Beginning download...")

    with open(args.out, "w", encoding="utf-8") as f:
        for i, oid in enumerate(ids, 1):
            record = fetch_with_retries(sickle, oid, "oai_dc", 3, 1.0)
            if record:
                data = dc_record_to_row(record)
                f.write(json.dumps(data, ensure_ascii=False) + "\n")

            if i % 50 == 0:
                print(f"  Processed {i}/{len(ids)}...")

if __name__ == "__main__":
    main()
```

Annex B: Embedding Generator (generate_embeddings.py)

```
import os
import json
import requests
from tqdm import tqdm
from dotenv import load_dotenv

load_dotenv()

# Configuration
INPUT_FILE = "harvest.jsonl"
OUTPUT_FILE = "mspace_embeddings.json"
ENDPOINT = os.getenv("AZURE_OPENAI_ENDPOINT")
KEY = os.getenv("AZURE_OPENAI_KEY")
DEPLOYMENT = os.getenv("AZURE_EMBEDDING_DEPLOYMENT")
API_VERSION = os.getenv("AZURE_EMBEDDING_API_VERSION", "2023-05-15")

URL = f"{ENDPOINT}/openai/deployments/{DEPLOYMENT}/embeddings?api-
version={API_VERSION}"
HEADERS = {"Content-Type": "application/json", "api-key": KEY}

def get_embedding(text):
    """Sends text to Azure OpenAI and returns the vector."""
    payload = {"input": text}
    try:
        resp = requests.post(URL, headers=HEADERS, json=payload)
        resp.raise_for_status()
        return resp.json()["data"][0]["embedding"]
    except Exception as e:
        print(f"Error generating embedding: {e}")
        return None

def main():
    processed_records = []
    # Read JSONL file
    with open(INPUT_FILE, "r", encoding="utf-8") as f:
        lines = f.readlines()

    print(f"Generating embeddings for {len(lines)} records...")

    for line in tqdm(lines):
        record = json.loads(line)
        # Create a rich context string for embedding
        content = f"{record.get('title', '')}. {record.get('subject', '')}. {record.get('description', '')}"

        # Generate vector
        vector = get_embedding(content[:8000]) # truncated to avoid token limits

        if vector:
            record["embedding"] = vector
            processed_records.append(record)

    # Save as a JSON array for Azure Search upload
    with open(OUTPUT_FILE, "w", encoding="utf-8") as f:
        json.dump(processed_records, f, indent=2)

    print(f"Saved {len(processed_records)} enriched records to {OUTPUT_FILE}")

if __name__ == "__main__":
    main()
```

Annex C: Azure Index Creator (recreate_index.py)

```
import os
import requests
from dotenv import load_dotenv

load_dotenv()

ENDPOINT = os.getenv("AZURE_SEARCH_ENDPOINT")
KEY = os.getenv("AZURE_SEARCH_KEY")
INDEX_NAME = os.getenv("AZURE_SEARCH_INDEX")
API_VERSION = "2024-07-01"

HEADERS = {"Content-Type": "application/json", "api-key": KEY}
URL = f"{ENDPOINT}/indexes/{INDEX_NAME}?api-version={API_VERSION}"

# Schema definition
index_schema = {
    "name": INDEX_NAME,
    "fields": [
        {"name": "identifier", "type": "Edm.String", "key": True, "searchable": False},
        {"name": "title", "type": "Edm.String", "searchable": True, "retrievable": True},
        {"name": "creator", "type": "Edm.String", "searchable": True, "retrievable": True},
        {"name": "subject", "type": "Edm.String", "searchable": True, "retrievable": True},
        {"name": "description", "type": "Edm.String", "searchable": True, "retrievable": True},
        {"name": "link", "type": "Edm.String", "searchable": False, "retrievable": True},
        {
            "name": "embedding",
            "type": "Collection(Edm.Single)",
            "dimensions": 1536,
            "vectorSearchProfile": "my-vector-profile"
        }
    ],
    "vectorSearch": {
        "algorithms": [
            {
                "name": "my-hnsw-config",
                "kind": "hnsw",
                "hnswParameters": {"metric": "cosine", "m": 4}
            }
        ],
        "profiles": [
            {
                "name": "my-vector-profile",
                "algorithm": "my-hnsw-config"
            }
        ]
    }
}

def main():
    # Delete existing index
    print(f"Resetting index: {INDEX_NAME}...")
    requests.delete(URL, headers=HEADERS)

    # Create new index
    resp = requests.put(URL, headers=HEADERS, json=index_schema)
    if resp.status_code == 201:
        print("Index created successfully with vector support.")
    else:
        print(f"Error creating index: {resp.text}")

if __name__ == "__main__":
    main()
```

Annex D: Data Uploader (upload_with_embeddings.py)

```
import os
import json
import base64
import requests
from dotenv import load_dotenv

load_dotenv()

ENDPOINT = os.getenv("AZURE_SEARCH_ENDPOINT")
KEY = os.getenv("AZURE_SEARCH_KEY")
INDEX_NAME = os.getenv("AZURE_SEARCH_INDEX")
API_VERSION = "2024-07-01"

URL = f"{ENDPOINT}/indexes/{INDEX_NAME}/docs/index?api-version={API_VERSION}"
HEADERS = {"Content-Type": "application/json", "api-key": KEY}

def main():
    with open("mspace_embeddings.json", "r") as f:
        records = json.load(f)

    # Transform for upload (Base64 encode keys)
    batch = []
    for rec in records:
        safe_id = base64.urlsafe_b64encode(rec["identifier"].encode()).decode()
        rec["identifier"] = safe_id
        rec["@search.action"] = "upload"
        batch.append(rec)

    print(f"Uploading {len(batch)} records...")

    # Upload in a single batch (add batching logic for >1000 records)
    payload = {"value": batch}
    resp = requests.post(URL, headers=HEADERS, json=payload)

    if resp.status_code == 200:
        print("Upload successful.")
    else:
        print(f"Upload failed: {resp.text}")

if __name__ == "__main__":
    main()
```

Annex E: Streamlit Application (streamlit_app.py)

```
import os
import base64
import requests
import streamlit as st
from dotenv import load_dotenv

load_dotenv()

# Configuration
SEARCH_ENDPOINT = os.getenv("AZURE_SEARCH_ENDPOINT")
SEARCH_KEY = os.getenv("AZURE_SEARCH_KEY")
SEARCH_INDEX = os.getenv("AZURE_SEARCH_INDEX")
OPENAI_ENDPOINT = os.getenv("AZURE_OPENAI_ENDPOINT")
OPENAI_KEY = os.getenv("AZURE_OPENAI_KEY")
CHAT_DEPLOYMENT = os.getenv("AZURE_OPENAI_DEPLOYMENT")
EMBED_DEPLOYMENT = os.getenv("AZURE_EMBEDDING_DEPLOYMENT")
API_VER = os.getenv("AZURE_OPENAI_API_VERSION", "2024-10-21")

st.set_page_config(page_title="Library AI Assistant", layout="wide")
st.title("📚 Library Search & Chatbot")

tabs = st.tabs(["🔎 Keyword Search", "💬 AI Chatbot"])

def decode_id(b64_id):
    try:
        return base64.urlsafe_b64decode(b64_id).decode()
    except:
        return b64_id

# --- Tab 1: Keyword Search ---
with tabs[0]:
    query = st.text_input("Search the collection:")
    if query:
        url = f"{SEARCH_ENDPOINT}/indexes/{SEARCH_INDEX}/docs/search?api-
version=2024-07-01"
        payload = {"search": query, "top": 5}
        headers = {"api-key": SEARCH_KEY, "Content-Type": "application/json"}

        resp = requests.post(url, headers=headers, json=payload)
        if resp.status_code == 200:
            for doc in resp.json().get("value", []):
                st.markdown(f"### {doc.get('title')}")
                st.write(doc.get('description'))
                link = doc.get('link', '#')
                st.markdown(f"[View Record]({link})")
            st.divider()

# --- Tab 2: RAG Chatbot ---
with tabs[1]:
    if "messages" not in st.session_state:
        st.session_state.messages = []

    for msg in st.session_state.messages:
        with st.chat_message(msg["role"]):
            st.write(msg["content"])

    if prompt := st.chat_input("Ask a question about the collection..."):
        st.session_state.messages.append({"role": "user", "content": prompt})
        with st.chat_message("user"):
            st.write(prompt)

        # 1. Embed Query
        embed_url =
f"{OPENAI_ENDPOINT}/openai/deployments/{EMBED_DEPLOYMENT}/embeddings?api-
version=2023-05-15"
        embed_resp = requests.post(embed_url, headers={"api-key": OPENAI_KEY},
json={"input": prompt})
        vector = embed_resp.json()["data"][0]["embedding"]

        # 2. Vector Search
        search_url = f"{SEARCH_ENDPOINT}/indexes/{SEARCH_INDEX}/docs/search?api-
version=2024-07-01"
        vector_payload = {
            "search": "",
            "vectorQueries": [{
                "kind": "vector",
                "vector": vector,
                "fields": "embedding",
                "k": 3
            }]
        }
        search_resp = requests.post(search_url, headers={"api-key": SEARCH_KEY},
json=vector_payload)

        # 3. Construct Context
        context_text = ""
        for doc in search_resp.json().get("value", []):
            context_text += f"Title: {doc['title']}\nDescription: {doc['description']}\n\n"

        # 4. Generate Answer
    chat_url =
f"{OPENAI_ENDPOINT}/openai/deployments/{CHAT_DEPLOYMENT}/chat/completions?ap
i-version={API_VER}"
    system_prompt = f"You are a library assistant. Answer using this
context:\n\n{context_text}"

    chat_payload = {
      "messages": [
        {"role": "system", "content": system_prompt},
        {"role": "user", "content": prompt}
      ],
      "temperature": 0.5
    }

    gpt_resp = requests.post(chat_url, headers={"api-key": OPENAI_KEY},
json=chat_payload)
    answer = gpt_resp.json()["choices"][0]["message"]["content"]

    with st.chat_message("assistant"):
      st.write(answer)
    st.session_state.messages.append({"role": "assistant", "content": answer})
```

About the author

Wei Xuan serves as the Associate University Librarian at the University of Manitoba Libraries, where he provides strategic leadership for projects centered on system interoperability, research support services, and enhanced accessibility. An active leader in the broader library community, he currently serves as the President of the CALA Canadian Chapter and is a member of the IGeLU Steering Committee. His research interests focus on data management, service evaluation, and the implementation of emerging technologies.